CS3025 Compiladores

Laboratorio 11.1 – 23 octubre 2023

Analisis Sintactico, Analisis Semantico e interpretes

1 Expresiones Regulares

1.1 Enteros Binarios (base 2)

a) Escribir una expresión regular que describa el lenguaje de números binarios. Números binarios con ceros a la izquierda son válidos.
b) Dada una cadena reconocida por la expresión regular definida en a), implementar la función bin2int que convierta a la cadena en el numero entero equivalente.

1.2 Enteros (base 10)

a) Escribir una expresión regular que describa el lenguaje de los números enteros. Los números enteros validos serán secuencias de dígitos sin ceros a la izquierda, con la excepción del número 0. Por ejemplo, 0,12 y 300 son números válidos, pero 00, 012, 00300 no lo son.
b) Dada una cadena reconocida por la expresión regular definida en a), implementar la función string2int que convierta a la cadena en el número entero equivalente

1.3 Reales (base 16)

a) Escribir una expresión regular que describa el lenguaje de los números reales de tal manera que solo números con dígitos en ambos lados del punto decimal son validos. Ademas no se aceptaran números con ceros a la izquierda e.g 12.3, 0.5, 0.0 son validos pero 012.3, 12, y .3 no los son.
b) Dada una cadena reconocida por la expresión regular definida en a), implementar la función string2float que convierta a la cadena en el número de tipo float equivalente.

1.4 Hexadecimales

a) El sistema hexadecimal es un sistema de numeración escrito en base 16. Los números hexadecimales usan 16 dígitos: los dígitos habituales y los caracteres A, B, C ,D ,E y F para representar los números del 10 al 15. Así, por ejemplo, A, 2B, 42, F1 y 100 denotan a los números 10, 43, 66, 241 y 256 en base 10, respectivamente. Escribir una expresión regular que describa el lenguaje de los números hexadecimales.
b) Dada una cadena reconocida por la expresión regular definida en a), implementar la función hex2float que convierta a la cadena en el número entero en base 10 equivalente.

1.5 Identificadores

Los identificadores de un lenguaje X empiezan siempre con un carácter alfanumérico seguido de una secuencia (de longitud >= 0) de caracteres alfanuméricos, numéricos y el carácter ´_´. El carácter '_' puede aparecer maximo una vez, en cualquier posición excepto al final de la cadena. Escribir la expresión regular y el AFD que .aceptan dichos identificadores.

2 Análisis Sintáctico

Dado un lenguaje con expresiones definidas por la siguiente sintaxis (concreta):

```
Exp ::= CExp      CExp ::= AExp (('<'|'<='|'==') AExp)?
AExp ::= Term ((´+´ | ´-´) Term)*   Term ::= Factor ((´*´|´/´) Factor)*
Factor ::= num | ´(´ Exp ´)´ | id
```

a) ¿Las cadenas 'x' y '255' pertenecen a Exp?
b) Como se parsea la cadena `'x+y < a-b'`? Escribir el árbol sintáctico correspondiente.
c) Escribir la función `void parseCExp()`.

La sintaxis abstracta de Exp puede representarse de la siguiente manera:

```
Exp ::= BinaryExp(Exp,Exp,op) | Num(n) | ID(x) | ParenthExp(Exp).  Op =
{plus, …, lt, leq, eq }
```

d) Asumiendo que existe un constructor por cada clase de la sintaxis abstracta (AST), agregar acciones semánticas a `parseCExp()` para generar el objeto correspondiente del AST.
e) Escribir el árbol sintactico **abstracto** de `'x+y<a-b'`.
f) Agregar a la sintaxis la clase sintáctica `BExp` que permite escribir expresiones booleanas con los operadores `and` y `or`. Estos operadores deberán tener la menor precedencia de todos. Por ejemplo, `x and y + 2` deberá agruparse como x and (y+2). La gramática deberá continuar siendo parseable usando el método de descenso recursivo. Además, deberá escribirse de tal manera que permita asociatividad a la izquierda.
g) Re-escribir la gramática de `BExp` para que permita asociatividad a la derecha.
h) Escribir la función `parseBExp` , una versión con asociatividad a la derecha y otra a la izquiera.
i) Escribir la funcion `parseBExp` con acciones semánticas para generar ASTs. ¿Necesitamos agregar alguna clase la sintaxis abstracta?
j) Dado que estamos trabajando con expresiones booleanas, podemos agregar las constantes `true` y `false`. ¿A que parte de la gramática pertenecen?
k) Queremos agregar los operadores `not` y `-`. Estos deberán tener mayor precedencia que el resto de operadores de tal manera que `'not x and x < y'` sea analizado como `'(not x) and (x<y)'`. Agregar la clase sintactica `Unary` a la gramática extendida:
   `Unary ::= …`
   para tomar en cuenta los nuevos operadores.
l) Implementar parseUnary(). Como debemos modificar la sintaxis abstracta y paseUnary() para poder generar ASTs?

3 Análisis Sintáctico y Semántico

3.1 Analisis Semanticos de BinaryExp

a) Especificar las reglas de verificación de tipos de BinaryExp dada la nueva gramática que include BExp, es decir, la gramática que incluye expresiones booelans con los operadores and y or, usando tcheck.
b) Especificar las reglas de verificación de tipo de UnaryExp suando tcheck.

3.2 do-while

Dada la sintaxis de expresiones y sentencias de IMP:

```
Stm ::= id "=" Exp |
        "print" "(" Exp ")" |
      "while" Exp "do" Body "endwhile" | …
Exp ::= BExp
….
```

a) Decidir y agregar la sintaxis de `do-while` a la gramática de tal manera que sea posible la implementación del análisis sintáctico que usa descenso recursivo.
b) Definir las reglas de verificación de tipos de do-while usando tcheck. Implementar en imp_typechecker.cpp.

3.2 Arreglos

Dada la sintaxis:

```
Program ::= Body
Body ::= VarDecList StmList
VarDecList ::= (VarDec)*
VarDec ::= "var" Type VarList ";"
Type ::= id
VarList :: id ("," id)*
StmList ::= Stm (";" Stm)* …
Stm ::=   id "=" Exp |
          "print" "(" Exp ")" |
          "if" Exp "then" Body ["else" Body] "endif" |
          "while" Exp "do" Body "endwhile" |
Exp ::= …
Factor ::= …
```

¿Como implementar arreglos?